CMSC417 Final Project

Tien Vu, Alexandra Maric, Anders Spear, Paul Beltran

List of Supported Features

- Being awesome by the virtue of being written in Rust.
- Fully downloading Art of War and Debian :)
- Communicating with tracker
- Reading a multifile `.torrent` file
- Talking to peers
- Downloading a full file (sometimes) (and not extremely slowly)
- Seeding to other peers

Design and implementation choices that you made

- Did not use threads at all, everything works through Rust’s Mio crate (epoll in rust).
- Split code into “modules,” with `main.rs` holding control over all state variables and passing them around as necessary.
- Each module performs different tasks (i.e. sending/receiving, tracker communication).
- All peer sockets are non-blocking.
- Instead of randomly targeting pieces, we complete the file from the first piece to the last. This is not ideal, but a little easier to manage which pieces we have yet to ask for and which pieces we should next ask for.
- We ignore the concept of “niceness” for the sake of simplicity.
- We always unchoke everyone and respond to everyone’s requests for the sake of simplicity.

Problems Encountered (And how they were addressed)

- OBOE (the code was actually right, and fixing the “error” made me [Anders] ignore the real error)
- Invalid requests causing peers to close the connection immediately. This was solved by examining what qBitTorrent sent as requests and emulating what it sent.
- Lack of input validation causing rogue peers to crash the program. This was an easy fix.
- A lot of unexpected internal behavior caused by complicated APIs, such as queued messages never getting cleared and causing excessive sends. Very difficult to track down and solve.
- Rust borrow checker. Solved by getting good.
- Rust errors. Solved by a lot of reading.
- Using Rust. Solved by crying.

Known Bugs or Issues

- yes!
- if you `dig` too deep im `main`ly `panic`ing
- Mild performance issues caused by the request strategy resulting in a huge chunk of time spent sending requests, receiving pieces, and then hashing in that order. The hashing portion causes a spike in CPU usage which is audible due to computer fans.
- On the topic of performance issues, we spend a lot of execution time accessing bitfields i.e. we have a small overreliance on using the BitVec crate to store information that we will access. This is likely due to `file.rs` checking if a piece is finished after every write by iterating over every byte in the block.
- We do not disconnect from seeders. In fact, we don’t even differentiate from seeders and leechers. We just connect to everyone and give pieces out like candy.
- There is no tracking of peer data uploaded/downloaded i.e. we have no notion of niceness. Someone could have a very slow connection that we are relying on for several blocks and we will still wait for their piece to come in (unless they do not finish within a timeout, like 30 seconds, otherwise we will request randomly from someone else).

Contributions Made

Tien

- `file.rs`: File writing, reading, management, hashing.
- `peers.rs`: Peer storage and management.
- Bugfixing galore.
- Worked with Alexandra to bug-smash and get the file to download :)

Anders

- `torrent.rs`: Parse `.torrent` file, make hashes and piece info avaliable to others.
- `p2p.rs`: Peer to peer communication, sends and receives data according to bt protocol.
- Pretty progress bar (its supposed to be blue but that part broke)

- Considerably less bugfixing.

Alexandra

- `main.rs`: Main event loop, timers, coordinates the other modules
- `strategy.rs`: Decides what messages to send
- Bugfixing galore part 2.
- Implemented seeder mode to test seeding

Paul

- `tracker.rs`: talks to the tracker with http
- Ideas man